**Project #1 Buffer Overflow Spring 2019**

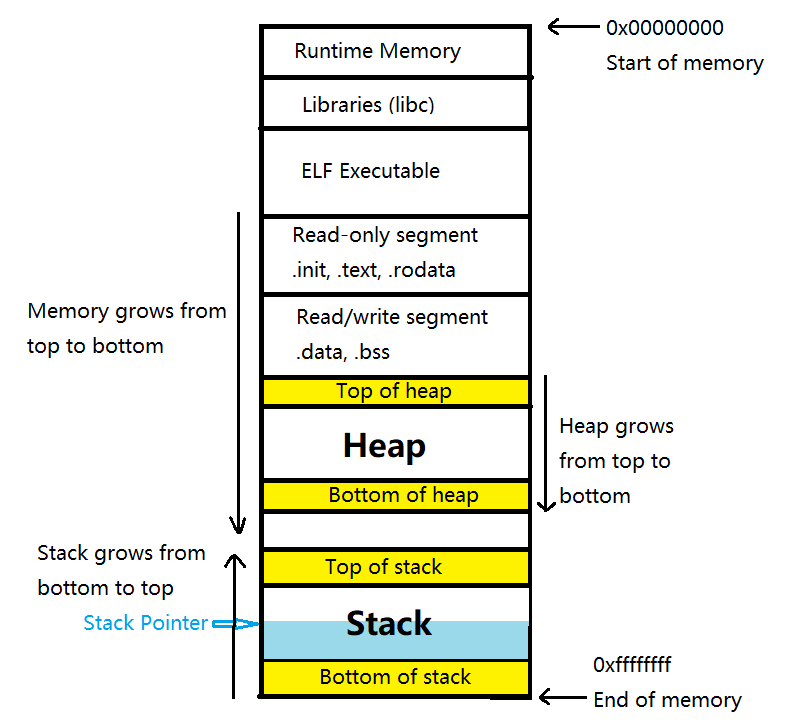
Name: CHANG Yingshan

Student ID: 903457645

GT Account: ychang363

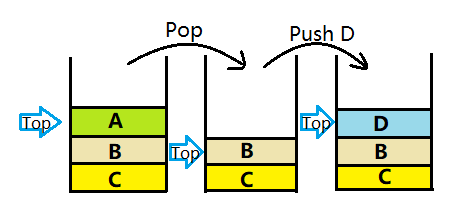
Understanding Buffer Overflow

1. **Stack buffer overflow**
   1. **Memory architecture**
      1. Location of stack in memory.

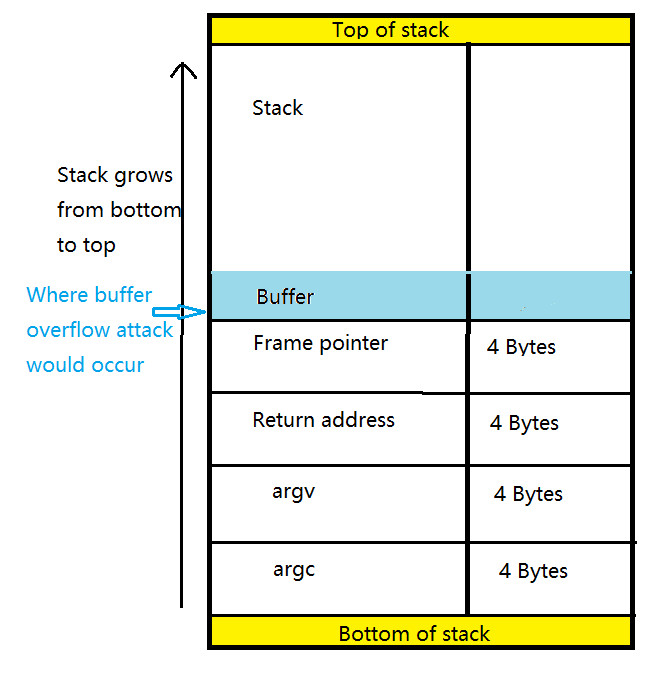


Stack is a special region of the computer’s memory that stores temporary variables created by each function. Once a stack variable is freed (popped), that region of memory becomes available for other stack variables. Stack is managed and optimized by the CPU efficiently. You don’t have to allocate memory by hand, or free it once you don’t need it.

* + 1. For example, initially the stack has three variables: A,B,C. When pop() is executed, the top element in the stack, which is A, will be popped off the stack. When push(D) is executed, element D will be pushed onto the stack and become the top element.



* + 1. When running a program, all information of function calls is stored in the stack. The stack structure is described in the following picture. Data related to a particular function call are pushed onto the stack in the order: arguments, return address, frame pointer and buffer (according to the size of local variables inside the function). The stack grows from bottom to top, which is the inverse direction relative to overall memory. If inside function a, another function b is called, then information of function b is pushed onto the stack (on top of information of function a). When execution of function b is finished, information of function b will be popped off the stack.



* + 1. Misalignment issue within a stack

Each data type has alignment requirement. Typically, on a 32-bit machine, the processing word size will be 4 bytes. Thus, if an integer of 4 bytes is allocated on X address (X is a multiple of 4), the memory will read all 4 bytes of an integer in one memory cycle. However, if the integer is allocated at an address other than multiple of 4, it will span across two rows and require the memory read 2 cycles to fetch the data.

Naturally every member of structure should be aligned because of the alignment requirement. Misalignment will be solved by padding. For instance:

//Definition of a structure

typedef struct aaa

{

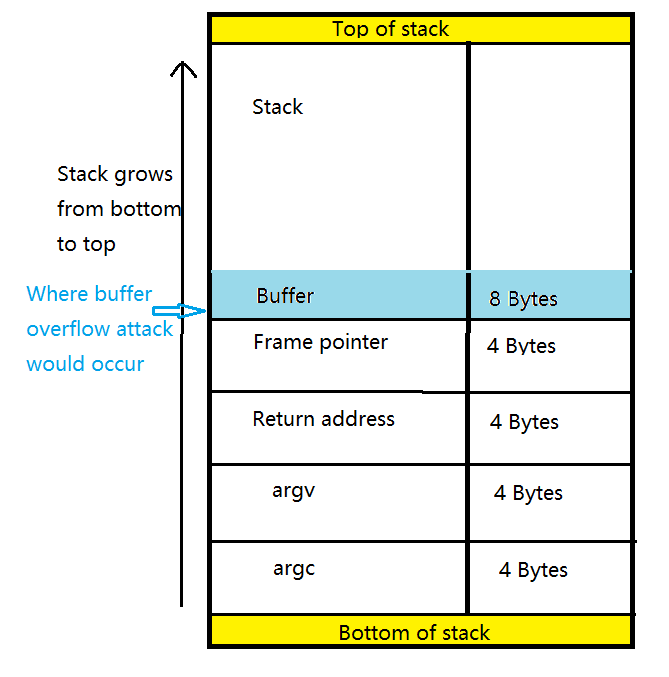
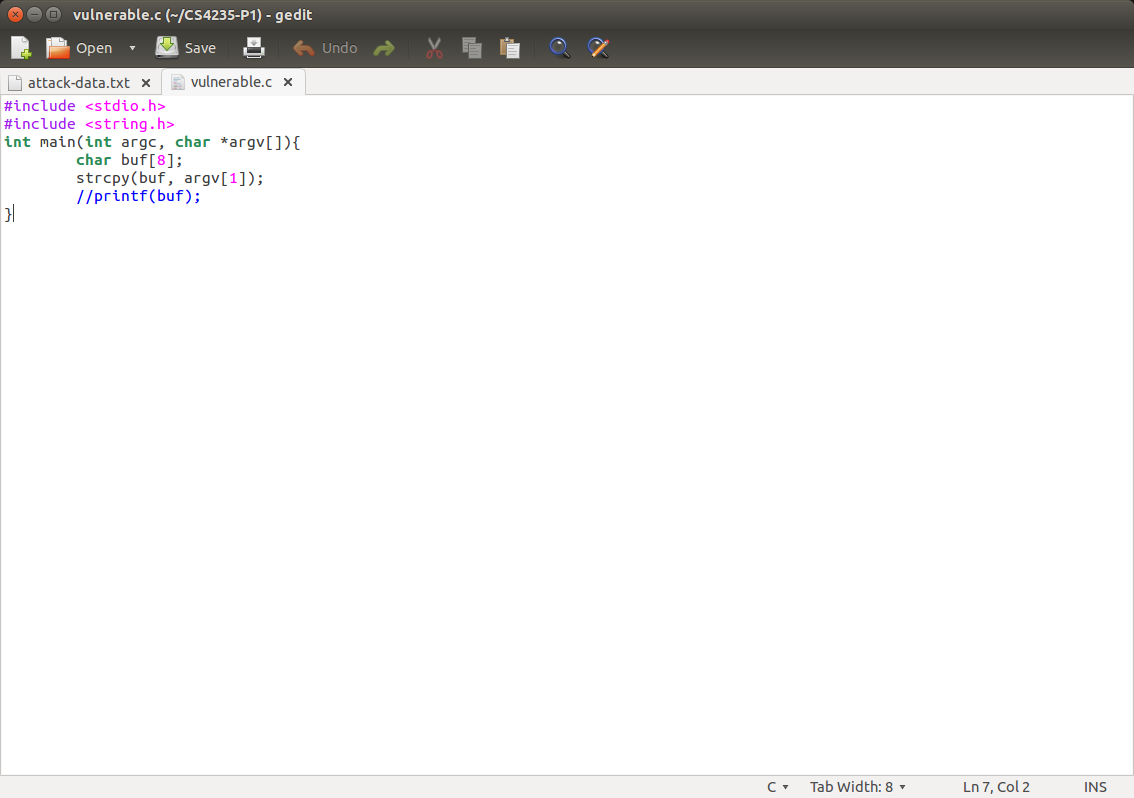
short int s;

char c;

} aaa;

The first member of aaa is char, which occupies 1 byte. If the second element of short int type is immediately allocated after the char element, it will start at an odd address. To avoid this, the compiler will insert a padding byte after the char element to ensure that short int element will have a starting address which is multiple of 2.

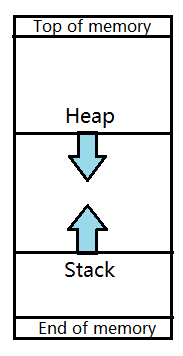
* 1. **Testing program that contains a stack buffer overflow vulnerability**



This is a vulnerable program. Buffer is allocated with 8 bytes in main(). If the input argument is more than 8 bytes, overflow is caused. The overflow direction is from top to bottom. In order to reach return address, we need to firstly overwrite all the memory allocated for buffer and frame pointer. Totally, the size of overflowing needed to fully overwrite return address is 16 bytes.

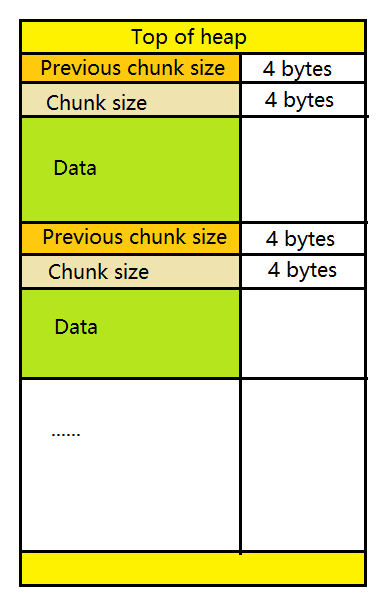
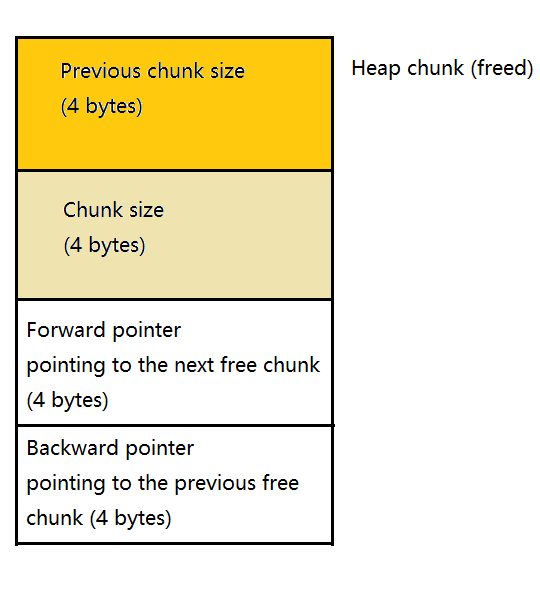
To exploit stack overflow, we want to overwrite the return address with the address of code that we want to be executed. We need to fulfill the buffer and frame pointer with data and make sure that the address of malicious code is exactly allocated in return address.

1. **Heap buffer overflow**
   1. **Memory architecture**



A picture of the whole memory architecture is shown in the previous question. Heap is located in the memory before stack and they are growing in opposite directions. Heap is growing from low memory to high memory, while stack is growing from high memory to low memory.

* 1. **Testing program that contains a heap buffer overflow vulnerability**
     1. Interpretation of heap chunk structure

Heap chunk structure depends on the current state of the chunk. For an allocated chunk, the metadata present are “previous chunk size” and “chunk size”, each field occupying 4 bytes. This means an allocated chunk always has 8 bytes of metadata, after which the actual buffer starts. The first bit of “chunk size” field indicates whether or not the previous chunk is in use. If it’s set, then the previous chunk is in use. If it’s not set the previous chunk is freed.

For a free chunk, we also have “fd” and “bk” fields, each needs 4 bytes of memory. Fd points to the previous free chunk, and bk point to the next free chunk. This means free chunks are saved in a doubly linked list. It is important to note that there isn’t just one linked list. In fact, there are multiple linked lists storing free chunks of different sizes, which makes searching for a free chunk with a specific size much faster.

* + 1. Vulnerable code

#include <stdlib.h>

#include <string.h>

int main( int argc, char \* argv[] )

{

char \* first, \* second;

/\*[1]\*/ a = malloc( 8 );

/\*[2]\*/ b = malloc( 8 );

if(argc!=1)

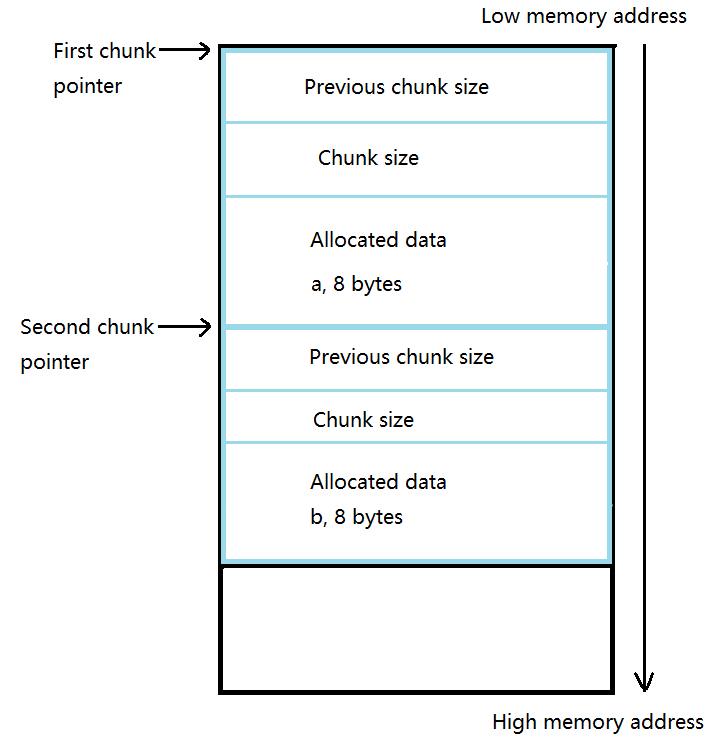
/\*[3]\*/ strcpy( a, argv[1] );

/\*[4]\*/ free( a );

/\*[5]\*/ free( b );

/\*[6]\*/ return( 0 );

}



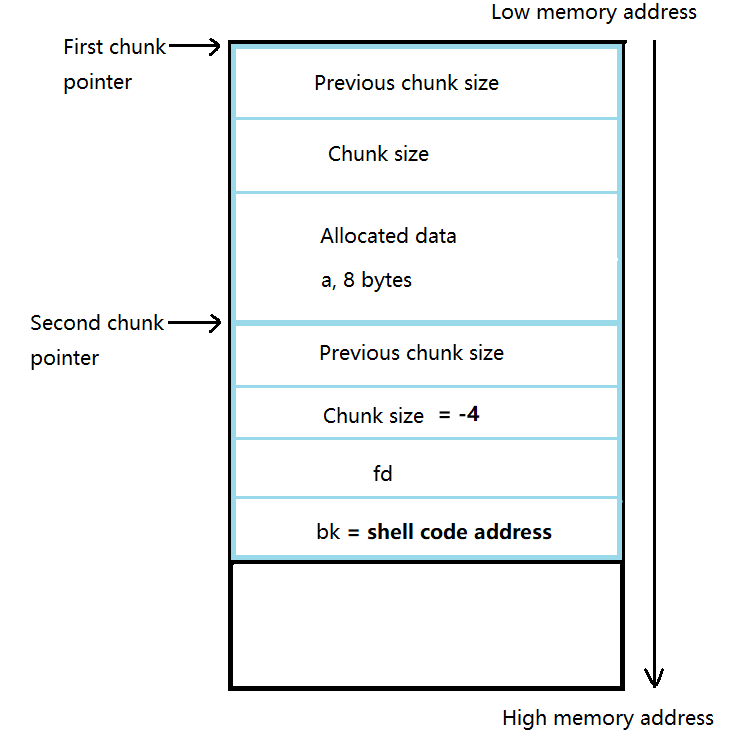
The two malloc() in line [1] and [2] allocated two chunks, each has a data field of 8 bytes.

* + 1. Heap exploitation

Fd and bk fields are the fields we can abuse in heap overflow exploitation.

Firstly, we need to understand how a chunk is freed. When a chunk is freed, it checks whether the chunks before and after it are free. In case the previous or next chunk is free, we can coalesce the free chunks together. Therefore, this increases the size of the free chunk. As a result, the free chunk has to be unlinked from the current linked list and be placed in another.

Secondly, the basic idea of exploitation is that in line [3], we overwrite the second chunk with shell code and trick the system to believe that the second chunk is already free when executing line [4]. In order to do this, we need to put a negative number in the “chunk size” field of the second chunk.



Then, when line [4] is executed, the system will check whether the next chunk is free. In order to do this, it will look at the first bit of “chunk size” field in the next-next chunk, which can be done by adding the second chunk’s size to the second chunk’s pointer. Since the attacker has overwritten the second chunk’s size with a negative number, the system will “look back” and “believe” that the first bit of “chunk size” field in the next-next chunk is unset, which means the next chunk is free.

Now the first and second chunk are considered as a “big free” chunk so the system will “unlink” the second chunk (since the system believes it is already free, the system thinks it is stored in a free chunk list) and place the “big chunk” into a free chunk linked list supporting the corresponding size. When the second chunk is “unlinked”, its bk will be copied to the bk field of its previous free chunk. BINGO! When this is invoked, the shell code stored in bk gets executed!

Exploiting buffer overflow

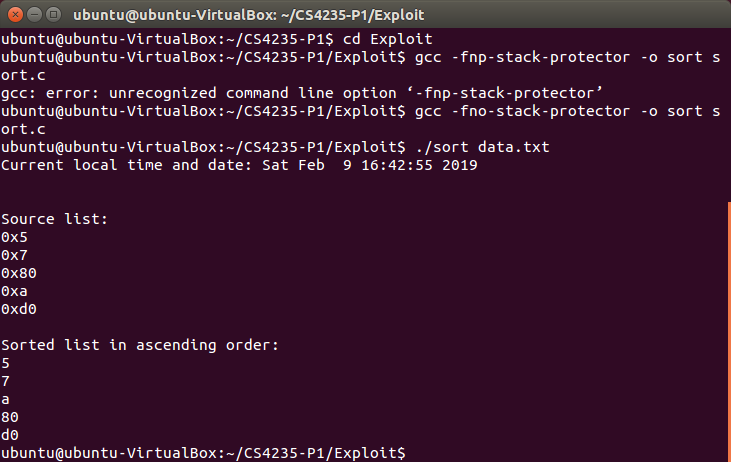
1. **Host machine and VM information**

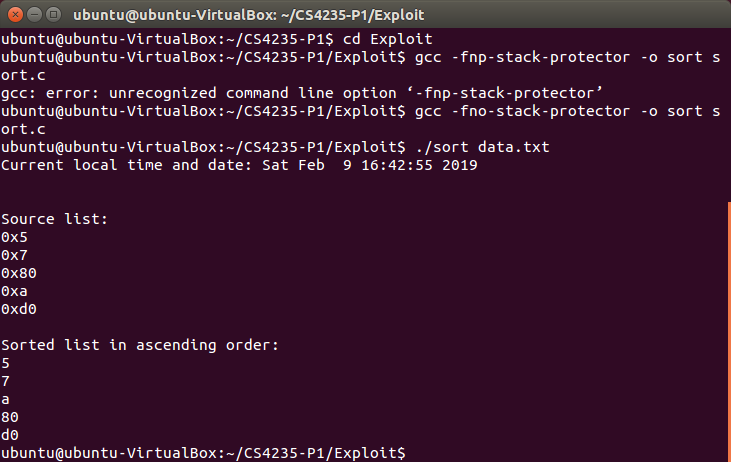
Host: PC, Windows 10, 64-bit

VM: Ubuntu 14.04.3, 32-bit

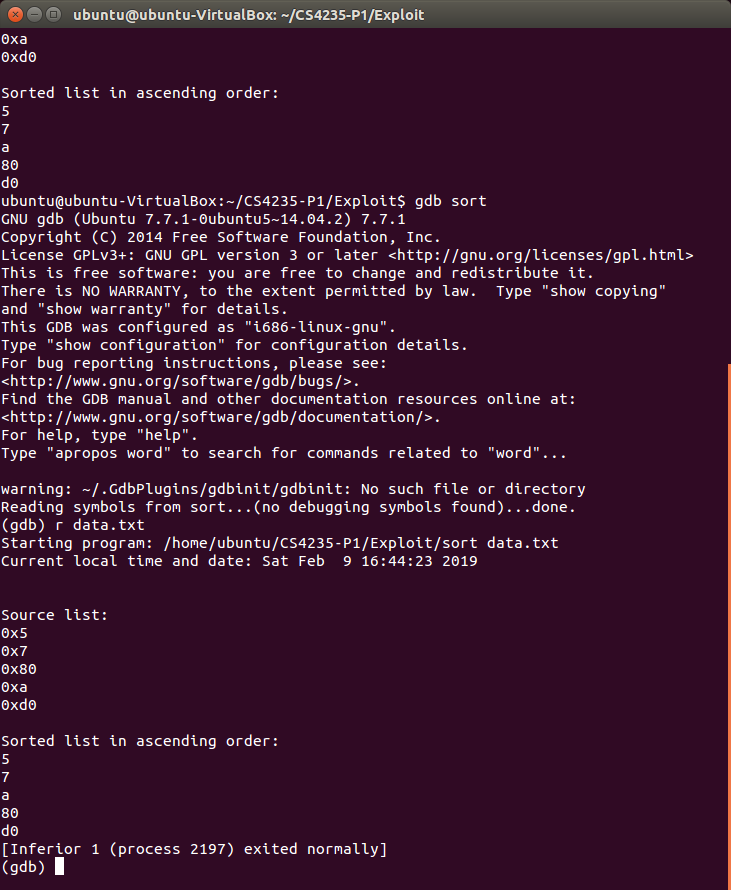


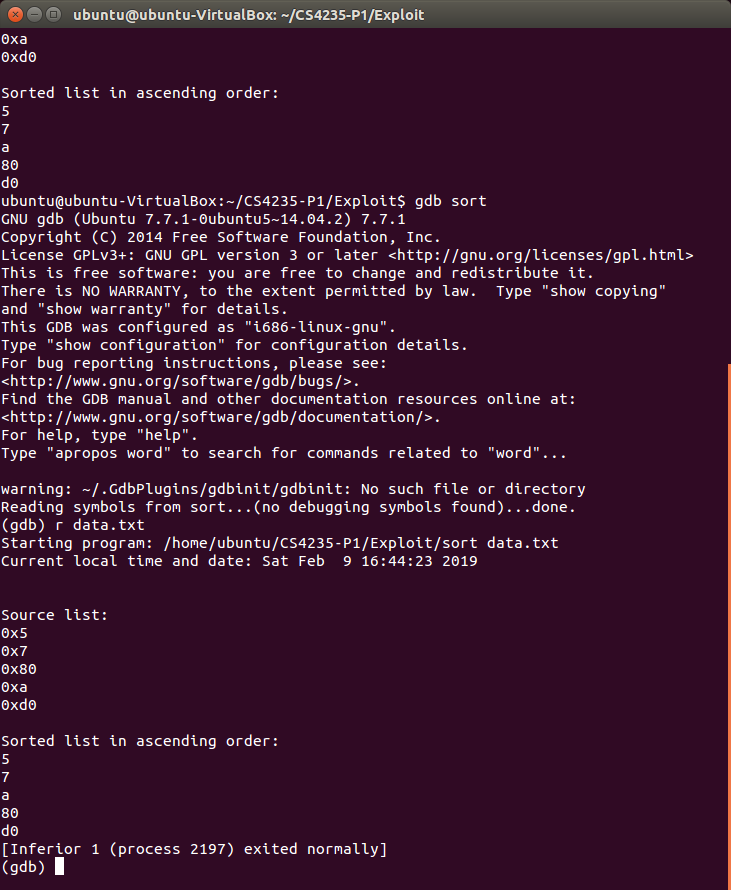
1. **Compile sort.c and show the normal output of the code**





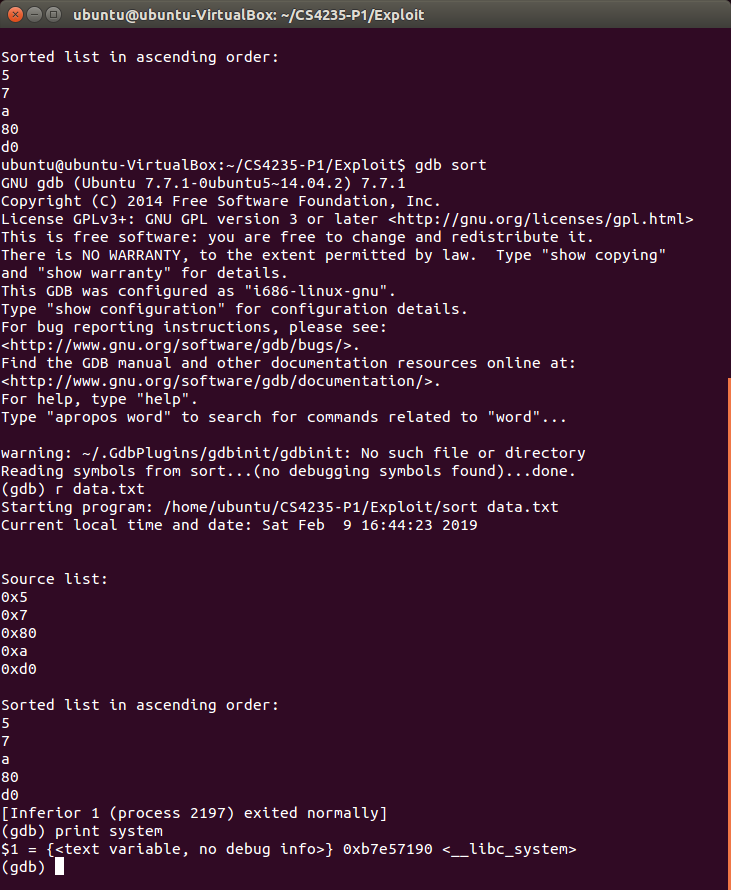
1. **Enter gdb in order to find the address of system() and sh**

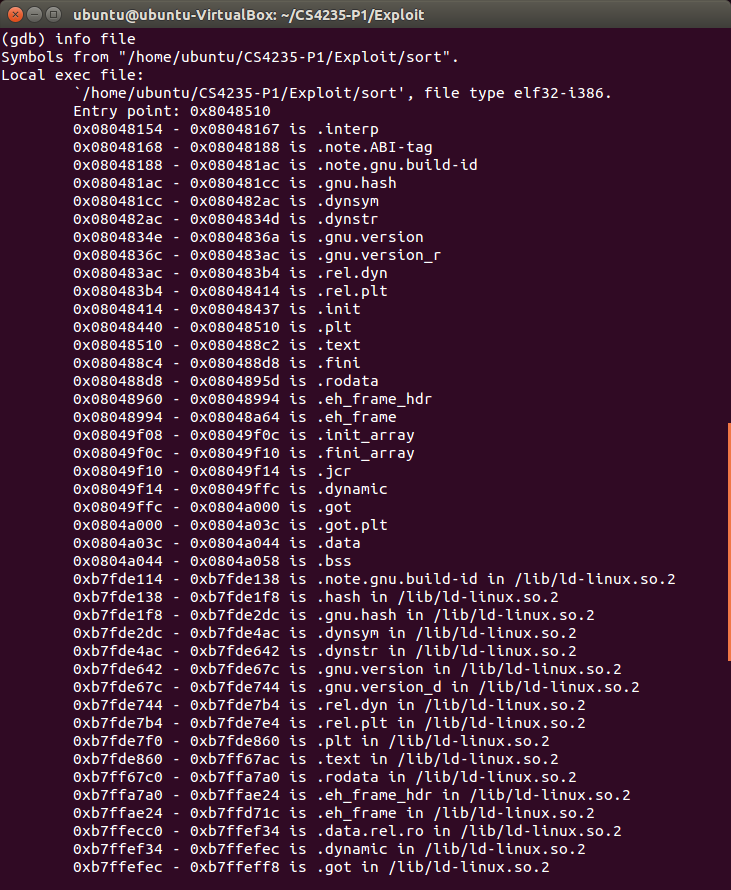


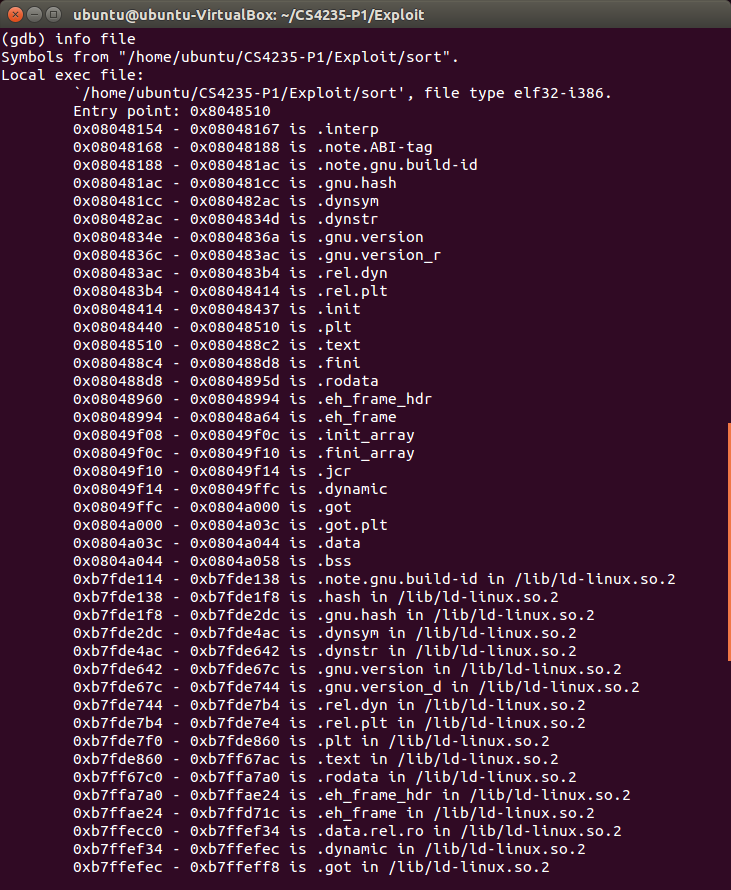


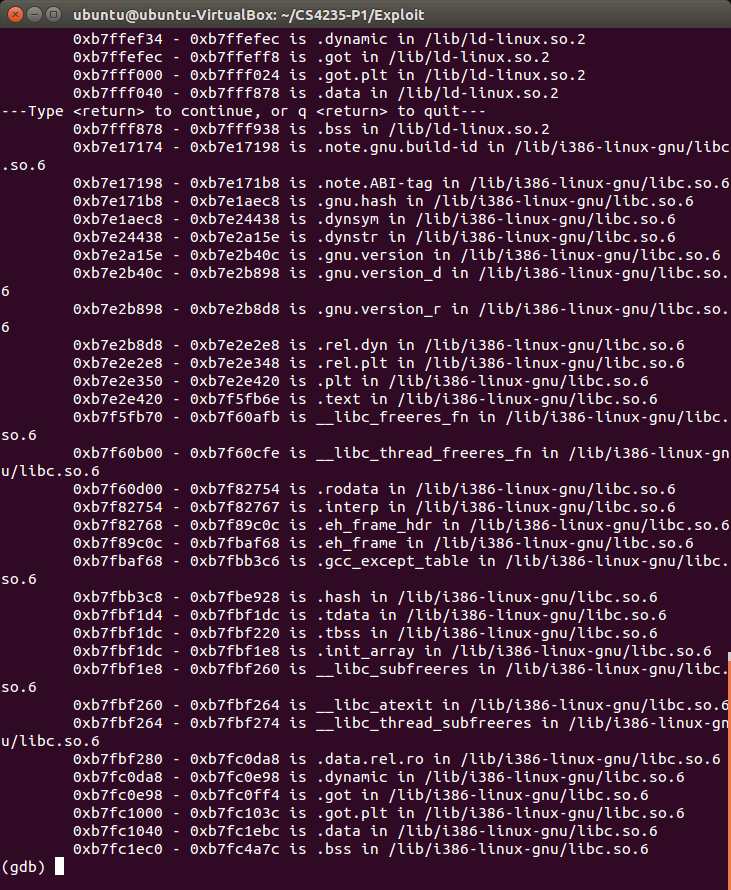
1. **Search for the address of system() and “sh”**

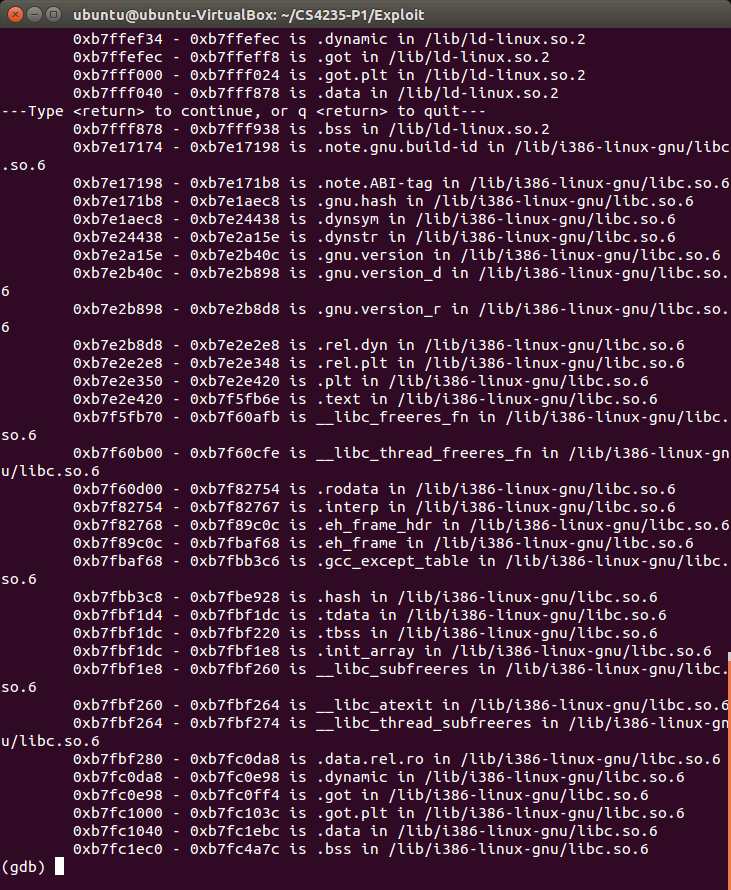
Address of system(): 0xb7e57190



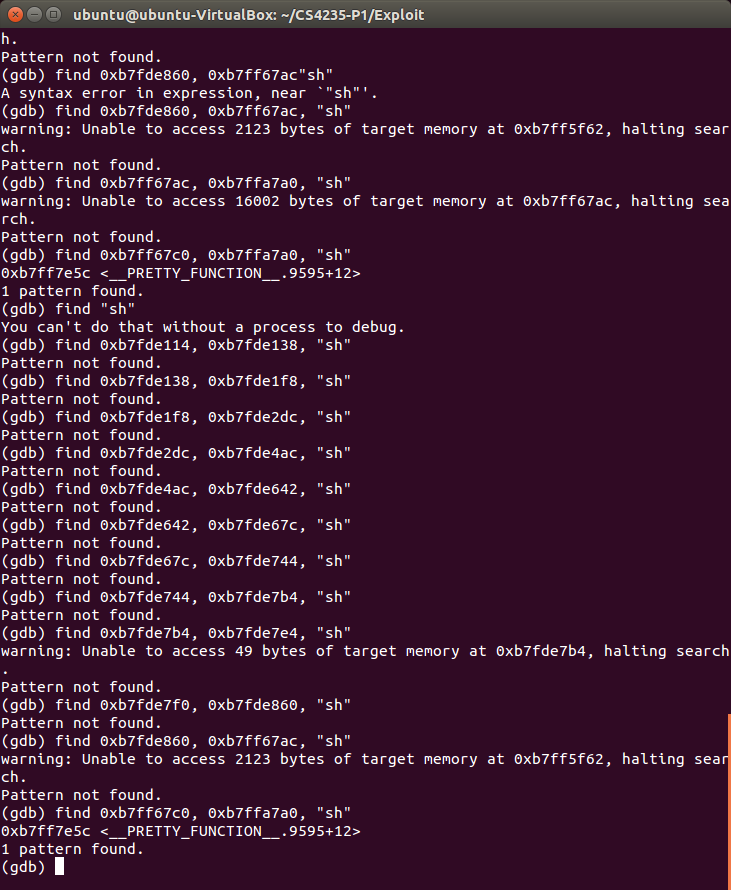








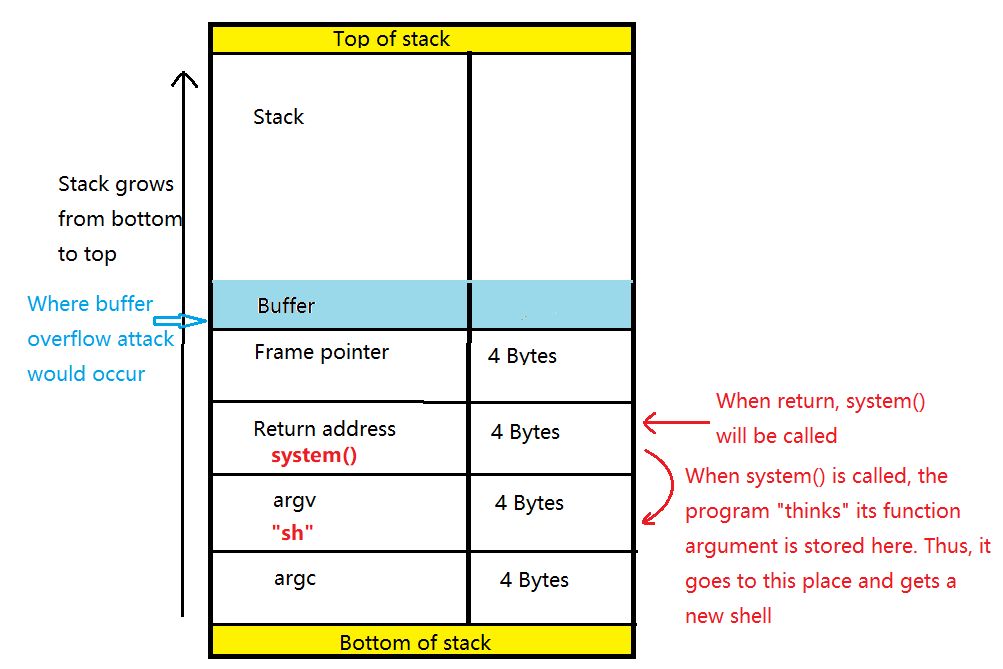
Next, search for the address where we can enter a shell code. The starting point of searching is the first mention of “lib”, which is more likely to be the place of “sh”. After searching through most of the memory, I finally found the address of “sh”.



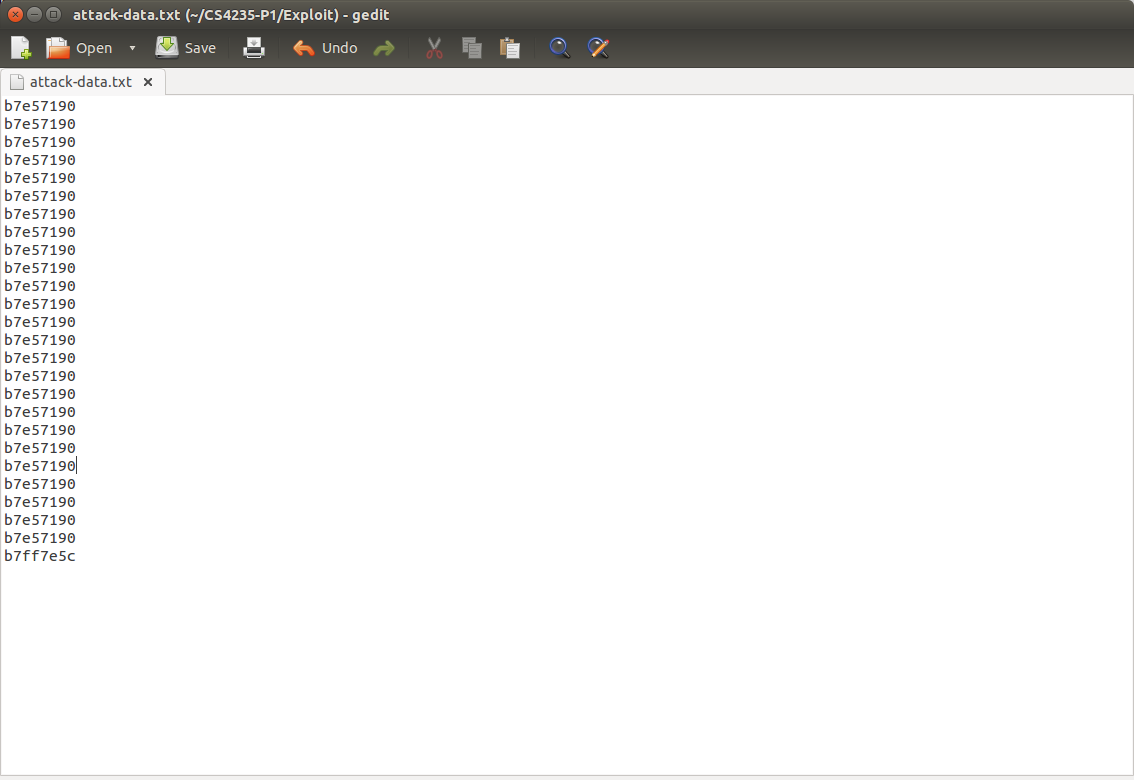
Address of “sh”: 0xb7ff7e5c

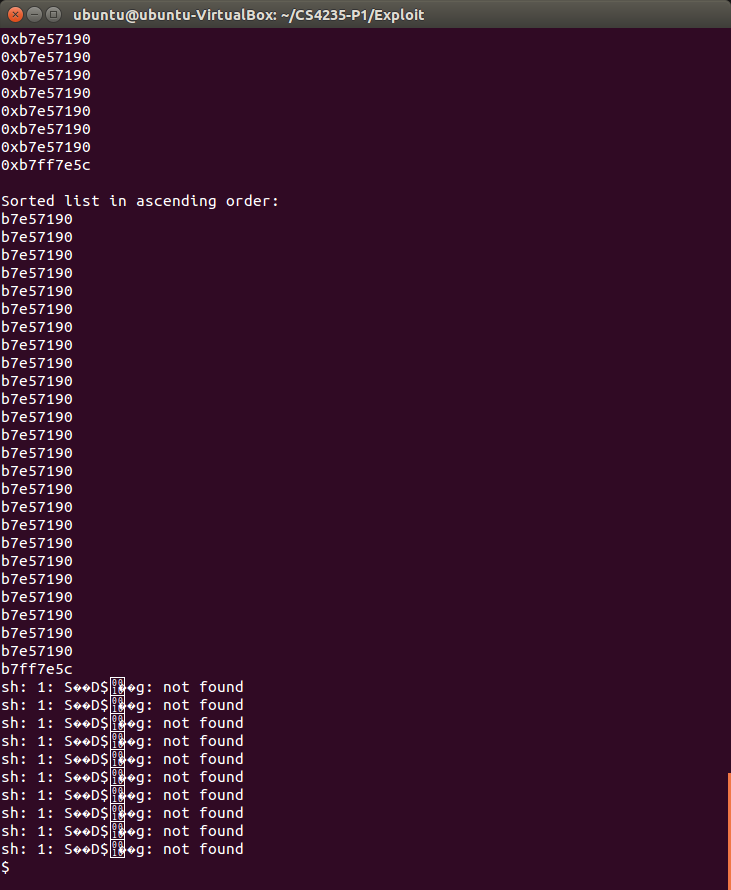
1. **Craft shell code in attack-data.txt**

I opened a new .txt file named “attack-data” and entered a long list to overwrite the return address with the address of “system()”.



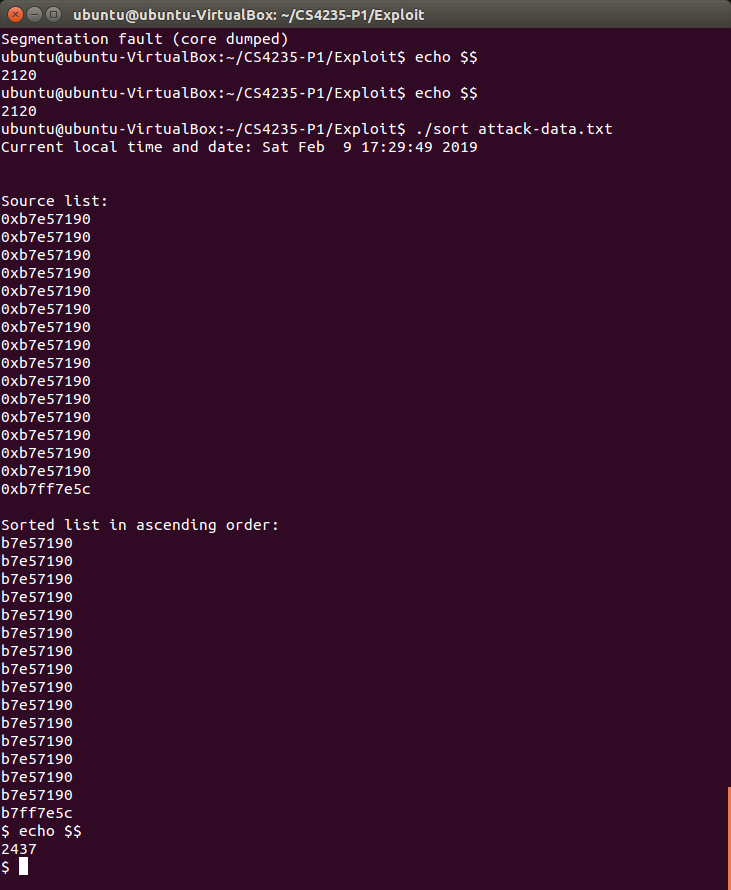
The key of successful exploitation is to overwrite all places in buffer and frame with anything, and then overwrite return address with “0xb7e57190” and argument with “0xb7ff7e5c”. In the first trial, I just copied a long list of “0xb7e57190” in order to fill in buffer and frame pointer, which looked like the following.



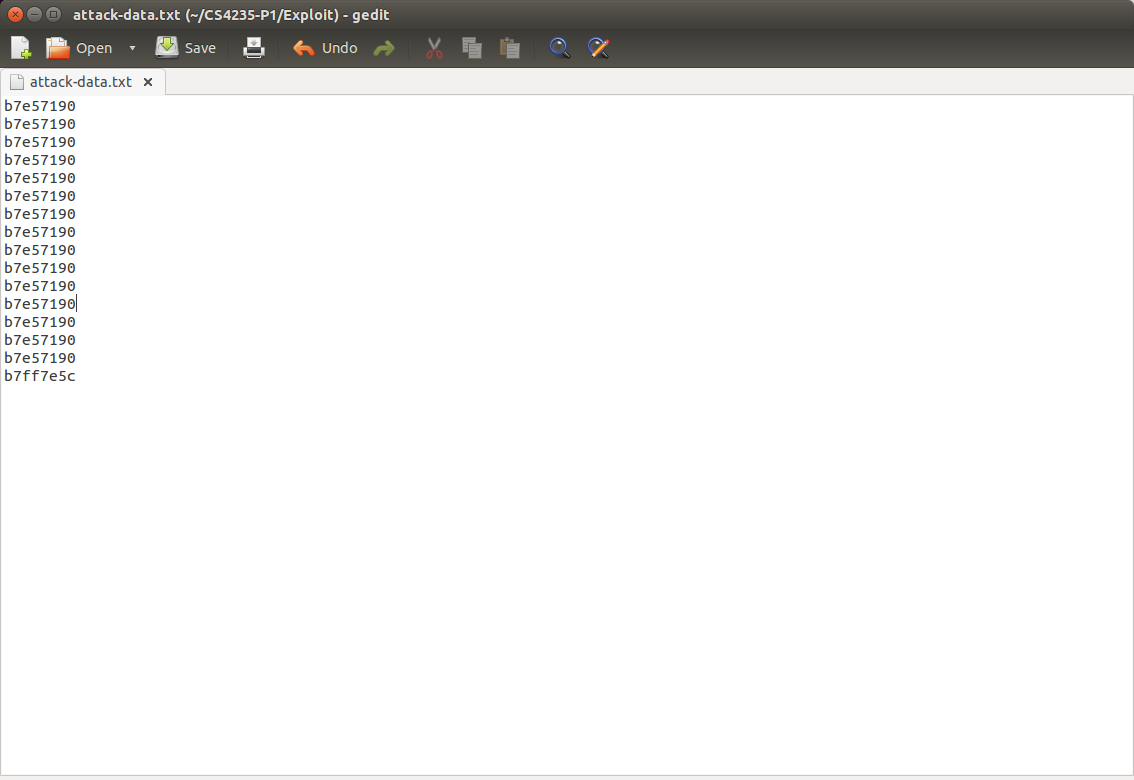


After execution of sort.c with input attack-data.txt, a new shell was opened, but before that there were ten lines of “sh: 1: S��D$��g: not found”. This means the overflow was longer than enough.

I deleted ten lines of “b7e57190” and we note that a new shell code was opened immediately with a different PID without segmentation fault.



The final attack-data.txt looks like this:



Reference

1. Stack overflow:

<https://gitea.simcop2387.info/dasvootz/GT_6035_CompSec/raw/commit/3160f307449c2e1fda15588d922d8b1e2814638a/project1/EricVootsProject1.pdf>

1. Heap overflow:

<https://sploitfun.wordpress.com/2015/02/26/heap-overflow-using-unlink/>

1. Struct member alignment and padding: <https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/>
2. Heap exploitation:

<http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf>